# Thesis

Xinyue Zhang

April 22, 2018

## Abstract

Formal verification is the process of checking whether a program satisfies some specification~~s~~. Dependent Haskell and F* are two languages that aim at awk program verification. However, the Haskell and F* communities generally conduct research separately and have rarely explored each other's approach.

This thesis compares Dependent Haskell and F* with a focus on their user interface and handling of nontermination. In terms of user interface, we collected examples to discuss their syntax, programming resources and proof complexity. As for the handling of nontermination, we demonstrated the trade-offs made by them and analyzed their strengths and limitations. Through our analysis, we conclude that F* has a more intuitive interface that comes with an elegant syntax and an engaging proof experience. Dependent Haskell, however, provides a better library and documentation support. Regarding to the handling of nontermination, Dependent Haskell does not have a termination checker but F* enforces terminating checking by default. Dependent Haskell supports verification of potentially divergent functions in full, regardless of being only weakly reliable (compile-time type check is insufficient for determining program correctness). On the other hand, F* is designed to be strongly reliable, but is currently lacking an applicable solution to verify potentially divergent functions. The thesis concludes with a detailed discussion on the preferable use cases of each language and some suggestions on future development directions for each community.

# 1 Conclusion

In this thesis we have presented and compared Dependent Haskell and F* focusing on two aspects: user interface and handling of nontermination.

Our discussion on user interface suggested that F* provides more readable syntax and a better proof support, but Dependent Haskell has more programming resources.

Singletons section vs verification techniques section

Specifically, Dependent Haskell's syntax is intricate with its introduction of singletons ? for historical reasons. However, it has many well-maintained libraries and well-supported documentation. Proofs in Dependent Haskell can be overwhelming, as every single step of a proof is expected. Programmers generally encode each theorem as a GADT and specify it in a separate lemma. On the other hand, as a newly-designed language, F* has an elegant syntax, supporting both dependent and refinement types. Nevertheless, it still lacks many library and documentation resources. In terms of theorem proving, F* provides an intuitive and semi-automated interface, abstracting over its type system and integrating with the Z3 solver. As such, it generally requires less code than Dependent Haskell and greatly reduces programmers' proof burden. It is worth pointing out that F*'s interface slowly approaches the complexity of that of Dependent Haskell, when verification goes beyond the ability of Z3.

this sentence does not make sense.

what does this mean?

As far as our analysis on handling of nontermination, we demonstrated the trade-offs made by each language. Without a termination checker, Dependent Haskell supports theorem proving on functions that may not terminate. Yet, Dependent Haskell is weakly reliable, as proofs on type equality require additional verification at run time to determine their correctness. F*, on the other hand, strictly enforces its type system to be strongly reliable. However, at this stage, it has not provided an applicable solution to verify functions with effect `Dv`.

From our analysis, we found F* to work better with small-to-medium sized verification of total functions due to its elegant syntax and user-friendly interface. For properties associated to functions that may not terminate, we suggest using Dependent Haskell instead. In terms of larger verification demands, we suppose both languages to provide about-equal support, but each has some limitations. Specifically, Dependent Haskell, without any proof automation, would result in enormous programs that are both difficult to program and hard to maintain. On the other hand, F* heavily depends on the Z3 SMT solver for verification. As a result, it can be very time consuming as the context presented to Z3 becomes larger.

you have no evidence to support this claim.

was this observed in practice?
GHC can also be slow.

At this stage, we believe that F* is more suitable for people new to program verification and Dependent Haskell is better for professional developers. Although F* provides a more enjoyable programming experience for its users, it has a complicated procedure to install and execute. It requires programmers to separately install the Z3 solver and the general F* compiler, and to manually set up all associated path variables. In addition, it is not directly executable. F* programs need to be extracted into either OCaml, F# or C for execution. On the other hand, Dependent Haskell, as an extension of Haskell, has a mature development process. It bundles all its resources, including the main GHC

F* compiler is an executable, no?

compiler, Cabal library installation tool, etc, into a package that can be directly installed. The biggest highlight worth mentioning is that all programs written in Dependent Haskell can be directly executed as a normal Haskell program. It's unclear what this sentence means.

Finally, we propose the following suggestions for development in each language: Firstly, Dependent Haskell might benefit from redesigning its syntax, especially focusing on finding an alternative to singletons. We believe that Eisenberg's complete work on full-fledged dependent types would be a good start, and we hope to see its implementation in GHC soon. Secondly, F* might consider improving its support for verification of `Dv` functions that may not terminate.

this ¶ adds little. You've identified shortcomings and propose to improve them, unsurprisingly.

As the Dependent Haskell and F* communities continue their development, we look forward to more and more collaborations between them. In the future, we would also like to continue examining the trade-offs made on handling nontermination. We are confident in both communities towards a language with a more intuitive interface and an improved support on proving potentially divergent functions.